FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

"Profiling"por hardware em tempo real para sistemas embebidos

Rui Alves

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: João Paulo de Castro Canas Ferreira

Co-Orientador: João Carlos Viegas Martins Bispo

17 de Julho de 2017

Resumo

O trabalho apresentado nesta dissertação contribui para responder à importância atual de acelerar sistemas embebidos. Para tal é necessário implementar um método antes da aceleração que descubra as zonas de código do programa com maior impacto no desempenho desse sistema. Até agora essa informação tem sido obtida por um programa a correr em modo offline. Para criar sistemas auto-adaptativos existe é importante adaptar essa aplicação para um módulo em hardware que possa ser colocado no sistema embebido total.

O módulo em questão tem como objetivo encontrar padrões chamados Megablocos, zonas com várias repetições de código no programa em execução. Estes padrões são compostos por blocos básicos, várias instruções com apenas um salto condicional. Após a deteção destes é possível procurar padrões para verificar se existem repetições. Nestas repetições podem encontrar-se o padrão Megabloco e a seguir, se o padrão for considerado adequado, guardá-lo. Para guardar o padrão é necessário guardar os blocos básicos que o compõem, e além disso a informação sobre tamanho deste as repetições e para que se possa mais tarde escolher que Megablocos acelerar em hardware.

Com o módulo concebido e avaliado neste trabalho obtiveram-se resultados comparáveis aos da aplicação de extração em software, identificando os mesmos Megablocos que esta, contando com diferenças insignificantes no número de repetições. Conclui-se também que o sistema consegue funcionar com a mesma frequência do processador Microblaze,100 Mhz.

Abstract

The work presented in this dissertation contributes to the current importante issue of accelerating embedded systems. To do this, it is necessary to implement a method before acceleration that discovers the code zones of the program with the greatest impact on the performance of that system. So far this information has been obtained by a program running in offline mode. To create self-adaptive systems it is important to adapt this application to a hardware module that can be placed in the autonmous embedded system.

The module in question aims to find patterns called Megablocks, zones with multiple code repetitions in an executing program. These patterns are composed of basic blocks, a linear sequence of several instructions with only one conditional jump. After detecting these it is possible to look for patterns to check if repetitions have been found. These repetitions constitute the desired Megablock pattern. If the pattern is considered appropriate it is saved to memory.

To save the pattern it is necessary to save the basic blocks that compose it, and also the information about the size and nunber of repetitions so that later it is possible to choose what Megablocks to accelerate in hardware.

With the module designed and evaluated in this work we obtained results comparable to those of the software extraction application, identifying the same Megablocks, with insignificant differences in the number of repetitions. The evaluation concluded by checking that the system can operate at the same frequency as the Microblaze processor, 100 Mhz.

Agradecimentos

A dissertação foi longa e com várias partes, por isso quero primeiro de tudo agradecer ao Prof. Dr. João Canas Ferreira pela a orientação ao longo da mesma, pela boa disposição com que me aconselhou e ajudou nessas várias fases.

Ao Dr. João Bispo quero também agradecer pelo tempo que disponibilizou a ajudar com certos conceitos e dúvidas. Da mesma forma quero agradecer ao Nuno Paulino pelas ajudas em alguns momentos desta dissertação.

Gostava de agradecer todos os meus companheiros que estiveram ao meu lado a trabalhar nas suas dissertações e que me deram motivação, por acreditarem na possibilidade de perfeição desta tese. Quero também agradecer a todos os meus amigos que me acompanharam ao longo de todo este percurso académico ou fora dele no trabalho e na folia.

Por último quero e devo de agradecer aos meus pais por me chatearem a cabeça sempre que as coisas correm menos bem e no entanto acreditarem em mim e me ajudarem a passar por todos os momentos da melhor forma possível.

Rui Miguel Almeida Alves

"Do you have the patience to wait till your mud settles and the water is clear?
Can you remain unmoving till the right action arises by itself?
The Master doesn't seek fulfillment."

Lao Tzu

Conteúdo

1 Introdução **1**
- 1.1 Contexto . . . 1
- 1.2 Motivação . . . 1
- 1.3 Objetivos . . . 2
- 1.4 Estrutura Dissertação . . . 2

2 Revisão Bibliográfica **3**
- 2.1 Hardware Profiling . . . 3
- 2.2 Megablocos . . . 5
  - 2.2.1 Blocos Básicos . . . 5
  - 2.2.2 Definição de Megabloco . . . 6
  - 2.2.3 Algoritmo de deteção . . . 6

3 Conceção e Implementação **9**
- 3.1 Requisitos . . . 9
- 3.2 Opções do projeto . . . 9
- 3.3 Sistema final . . . 11
  - 3.3.1 Basic Block Detector . . . 12
  - 3.3.2 Módulo Megablock Pattern_Detect . . . 13
  - 3.3.3 Trace Buffer . . . 19

4 Resultados **23**
- 4.1 Bancada de teste e resultados . . . 23
  - 4.1.1 Resultados funcionais . . . 24
- 4.2 Recursos após Implementação . . . 25
  - 4.2.1 Tempos . . . 26
  - 4.2.2 Estimativas de Potência . . . 27

5 Conclusões e Trabalho Futuro **29**

A Exemplo de um Megabloco complexo **31**

B Documentação do módulo Megablock Extractor **33**
- B.1 Sistema . . . 33
  - B.1.1 Funcionalidades . . . 33
  - B.1.2 Parâmetros . . . 34
  - B.1.3 Modo de utilização . . . 34

Referências **37**

Lista de Figuras

2.1 Saltos condicionais . . . 4
2.2 Sistema do LEAP *profiler* . . . 4
2.3 Sistema ProMem . . . 5
2.4 Código do Microblaze em assembly a representar um Megabloco . . . 6
2.5 Programa executado com blocos básicos, fragmentos e os Megablocos formados por estes . . . 7

3.1 Examplo do benchmark Motion_estimation-O2 . . . 10
3.2 Módulo de topo: Megabloco extrator . . . 11
3.3 Módulo Basic block detector . . . 12
3.4 Exemplo de um caso deteção do início e fim de um bloco básico sem *delay slot* . . . 13
3.5 Módulo Megablock pattern_detect . . . 14
3.6 Módulo Square_detector . . . 15
3.7 Diagrama temporal: Módulo Square_detector . . . 15
3.8 Módulo encoder . . . 16
3.9 Patter_control e Counter fsm . . . 18
3.10 Diagrama temporal das máquinas de estados . . . 20
3.11 Diagrama de blocos do módulo Trace buffer . . . 20
3.12 Diagrama temporal: Módulo MB memory . . . 21
3.13 Componentes de MB memory. . . . 21
3.14 Bloco Mb Memory. . . . 22

4.1 Testbench e sua consolidação com o DUT . . . 23

A.1 Megabloco exemplo maior pt1 . . . 31
A.2 Megabloco exemplo maior pt2 . . . 32

B.1 Módulo Megablock Extractor . . . 33
B.2 Exemplo de Megablocos registados em memória . . . 35
B.3 Exemplo da interface com a memória . . . 35

Lista de Tabelas

3.1 Interface Módulo BB_detect, Clock e Reset excluídos 12
3.2 Interface Módulo Square_detector, Clock e Reset excluídos 14
3.3 Interface Módulo Encoder, Clock e Reset excluídos 16
3.4 Interface Módulo Control, Clock e Reset excluídos 19
3.5 Transições das Máquinas de estado FSM e Counter 19
3.6 Interface Módulo Trace_buffer, Clock e Reset excluidos 21

4.1 Megablocos encontrados pela aplicação e o sistema em hardware para alguns benchmarks 25
4.2 Resultados em número de iterações de Megablocos para alguns benchmarks . . . 26
4.3 Recursos pós implementação 26
4.4 Potência consumida pelo sistema 27

B.1 Interface do módulo 34

Abreviaturas e Símbolos

BB	Bloco Básico
FF	Flip Flops
DUT	Device Under Test
FIFO	First in First out
FSM	Finite State Machine
FPGA	Field Programmable Gate Array
GB	Gigabytes
GPP	General Purpose Processor
LUT	Look Up Table
RTL	Register Transfer Logic
RPU	Re-configurable Processing Unit
sbb	short backward branch

Capítulo 1

Introdução

1.1 Contexto

Em sistemas embebidos é prática comum usar processadores dedicados para aceleração da execução de certos programas. Em termos de opções para acelerar, uma das mais recentes será através de aceleradores em hardware, criados a partir da informação extraída a partir das sequências de código executadas pela aplicação. Com esta informação, o acelerador em hardware consegue implementar cada sequência de instruções por circuitos lógicos. Um *profiling* do código em modo *offline* é uma desvantagem dos sistemas implementados até agora, pois ficam a necessitar de algo externo ao sistema para funcionar (como é o caso da aplicação de *profiling*).

Neste momento *profiling* e técnicas de aceleração são muitas vezes intrusivas e introduz um *overhead* na de execução do processador. Por isso um acelerador que consiga introduzir optimização de forma transparente no processo de execução é desejável. Obtém-se um sistema autónomo quando houver forma de encontrar as informações sobre a execução e fornecer essa informação ao acelerador.

1.2 Motivação

A motivação para o trabalho proposto é de tentar acabar com uma das limitações destes sistemas embebidos com aceleradores. Neste momento a solução existente não é implementada de forma *online* em hardware. O trabalho anterior é uma aplicação de software *offline* que implementa o *profiling* do código em execução e essa informação é passada ao gestor de aceleração de um *soft-processor*, que por si cria os aceleradores necessários para otimizar os *hot-spots* do código. O acelerador constitui uma mais-valia do sistema embebido ao acelerar várias das zonas de cada programa em execução. Sendo assim *profiling* feito num sistema em hardware de forma *online*, ou seja interligado com o *soft-processor* e adquirir a informação dele a um ritmo equivalente ao da sua capacidade de processamento, trará um contributo para a aceleração de sistemas embebidos.

1.3 Objetivos

Otimizar sistemas embebidos através de aceleradores em hardware é uma tendência atual. Uma abordagem a esse problema foi proposta por João Bispo [1]. Este encontrou uma forma de otimizar partes significativas do código em execução dum sistema embebido detetando as partes deste que são frequentemente utilizadas através de uma ferramenta *offline*. Com a informação gerada pela mesma é possível usar estes dados para gerar o acelerador necessário. O acelerador é criado numa Re-configurable Processing Unit, que usa a informação retirada da ferramenta de *profiling* para executar em hardware o processamento correspondente às instruções mais usadas.

Este projeto de dissertação tem como objetivo obter o mesmo nível de informação da ferramenta, que neste momento existe em modo *offline*, de extração de Megablocos através de um sistema em hardware que faça essa extração de Megablocos *online*. Este ferramenta permitira implementar a deteção de Megablocos em hardware, monitorizando as execuções do programa num *soft-processor* e obter a informação necessária a enviar ao sistema que gere o acelerador. Um dos desafios desta abordagem será obter um sistema que não utilize um número elevado de recursos, embora permita obter uma elevada quantidade de informação sobre a execução do sistema embebido. Enquanto a aplicação pode guardar 2 GB de informação sobre o sistema, tal não acontece no sistema embarcado onde a ordem de tamanho da memória disponível será da ordem dos Megabytes, no máximo.

1.4 Estrutura Dissertação

Esta dissertação é composta por cinco capítulos, abordando os vários aspetos do trabalho.

No capítulo 2 são apresentadas as várias implementações de *profiling* em hardware e os resultados por elas obtidos. Além disso são expostos a definição de Megabloco e o algoritmo de deteção.

No capítulo 3 expomos as considerações sobre a implementação do detetor de Megablocos, tal como a explicação para a implementação de cada um dos seus sub-módulos.

No capítulo 4 mostra-se a forma como se testou esta implementação, os resultados obtidos por esta (comparando com os encontrados pela aplicação) e os recursos utilizados pelo módulo.

Por fim, no capítulo 5 são tiradas conclusões sobre o trabalho realizado e são também apresentadas possíveis abordagens futuras.

Capítulo 2

Revisão Bibliográfica

2.1 Hardware Profiling

Existem atualmente vários tipos de extratores de perfis de hardware. Alguns deles são simples contadores de *loops* e outros são *profilers* estatísticos de ciclos de execução. A deteção de *loops* é uma das mais simples formas de *profiling*; no entanto um *profiler* poder ter também informações sobre outros eventos específicos. Uma vez que nestas estatísticas os *loops*[2] são geralmente as maiores regiões do ciclo de execução, uma solução proposta foi detetá -los em tempo de execução através da verificação de algumas instruções significativas, sendo estas as chamadas de funções ou sbb (*short backward branches*), especificamente saltos que têm um deslocamento negativo de 256 ou 64. A solução foi implementada por meio de uma arquitetura baseada em *cache* que simplesmente armazenava os *loops* executados com mais frequência.

O *profiler* Dasprof [3] [4] trabalha de forma dinâmica para monitorizar o barramento de instruções do microprocessador de modo não intrusivo e determina os endereços das instruções do tipo *sbb*. O detetor de *sbb* funciona como um detetor de *loop* frequentes, não importando se estão desenrolados ou não. A arquitetura consiste num controlador do *profiler* e na *cache*. Esta usa os *sbb* como identificação para as informações sobre as iterações na *cache*. Esta arquitetura precisa de ter uma saída de interrupções a partir do microprocessador que analise a execução do *sbb* . É afirmado que esta arquitetura atinge precisão em *profiling* de 90% em iterações, 97% em execuções e 97% em percentagem total de tempo de execução, de um processador ARM9 com uma área adicional de 11%.

Considerando ciclos, há uma implementação para detetar *loops* dentro de *loops* [5] e fazer o *profiling* das instruções. Estes seriam então usados para otimizar e ajustar o desempenho de uma CPU multicore. A solução serve para registar os *sbb* dentro dos *loops* e seu endereço de destino, de forma a encontrar certas chamadas a funções ou *loops* dentro de *loops* , como se vê na figura 2.1. Com esta informação formam a vista hierárquica das estruturas dos *loops*. Além disso, registam o número de instruções dentro do *loop* e as várias instruções de cada *loop*. Em termos de resultados foram encontradas instruções que constituem maior parte do ciclo de execução.

Outra implementação para *profiler* de hardware é LEAP [6]. Este *profiler* usa uma tabela de

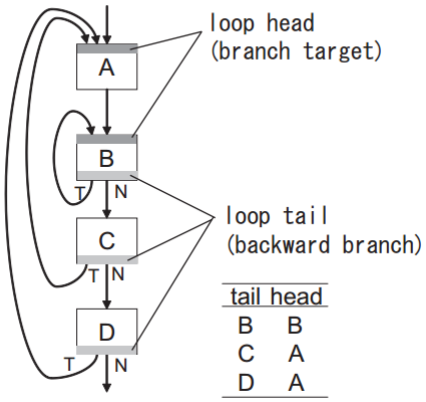

Figura 2.1: *Loop tails* (saltos condicionais para trás) e *loop heads* (objetivo para o salto) (fonte:[5])

dispersão para armazenar um contador de eventos, podendo estes ser chamadas de funções, *loops* ou outros. Considerando os eventos como funções, armazenar todas as instruções de função e o endereço que é executado levaria a uma sobrecarga de área grande. Devido a isso, esta técnica utiliza um hash perfeito que diminui a sobrecarga, produzindo um número de função com base no endereço de partida da função. Com isso pode guardar um índice exclusivo para cada função. O sistema funciona através de um monitor das instruções geradas pelo *soft processor* para saber se foi executada uma função ou *loop*. Com esta informação pode-se utilizar o Data Counter onde são incrementados os dados estatísticos sobre o sistema, que a seguir são passados ao Counter Storage onde é guardado toda essa informação (fig. 2.2). Esta técnica tem um problema no facto de que precisa extrair todos os endereços de funções do binário após a compilação do código, o que pode não ser viável em todas as situações. LEAP usa uma técnica de hashing perfeito para alcançar baixa sobrecarga de área e ter um de erro de precisão de 6% para vários *benchmarks*.

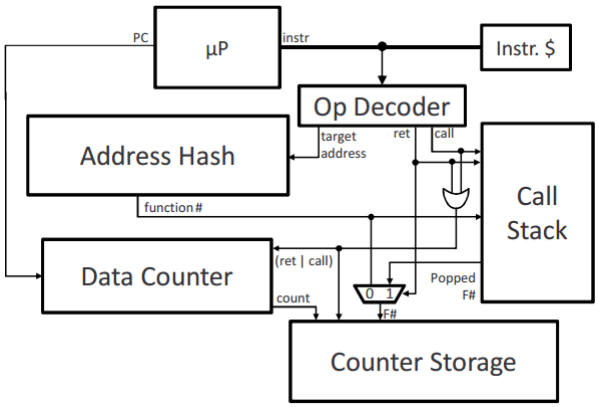

Figura 2.2: Diagrama de blocos do sistema do LEAP *profiler* (fonte:[6])

A monitorização do sistema é o objetivo das arquiteturas de *profiling*, mas Owl [7] não é simplesmente uma sonda de eventos, mas uma estrutura de monitorização programável que pode ser reconfigurada para monitorizar vários tipos de eventos. Essas estatísticas de eventos podem ser

usadas para correlacionar cada evento e reconfigurar a lógica para monitorizar a situação necessária, como por exemplo a geração de acesso à memória ou histogramas de faltas de *cache*.

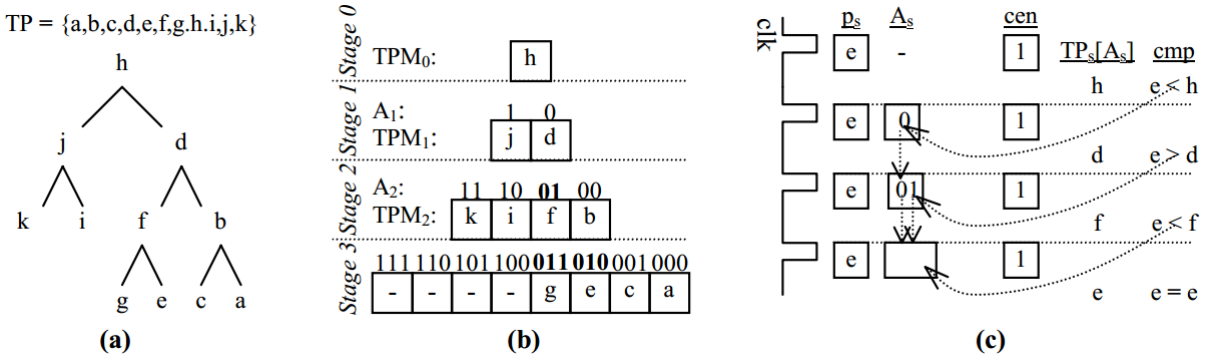

Figura 2.3: Sistema de *profiling* ProMem (fonte:[7])

Um *profiler* que funciona em tempo de execução deve ser capaz de processar a informação sem paragens. Para o conseguir o uso de encadeamento (*pipelining*) é recomendado. ProMem [8] é uma memória com arquitetura de árvore binária *pipelined*, como pode visto ser na figura 2.3, que pode monitorizar todo o tipo de instruções ou eventos. Para este sistema conseguir obter uma taxa de transferência de um padrão a cada ciclo, ele tem que fazer, em cada andar da árvore binária, uma comparação com o próximo bit da instrução. Para monitorizar o endereço de destino, existe uma arquitetura de memória que armazena todas as verificações de estado. Cada nó filho verifica o último bit de endereço da instrução e, se não encontrar uma correspondência, a instrução é atribuída ao próximo nó-filho até que cada nó-filho correspondente ao mesmo nó-pai seja verificado.

No caso de *profiling* e aceleração, Warp Processing [9] tem uma implementação interessante para otimização de sistemas embebidos mudando um kernel crítico do sistema para hardware. Isto é feito utilizando informação recolhida por um *profiler* dinâmico. O sistema transfere o código em binário para um processo de de-compilação complexo, onde o objetivo é encontrar as estruturas de alto nível, como os *loops*. Outra fase dessa compilação passa por manter operações complexas como a multiplicação que são transformadas em shift em código binário. Com esta abordagem é possível mapear o núcleo para a FPGA e melhorar o desempenho do processador.

2.2 Megablocos

2.2.1 Blocos Básicos

Um dos tipos de unidade usado para representar a estrutura da execução de um programa é o bloco básico. Estes blocos são um pedaço linear de código com várias instruções onde existe apenas uma saída condicional, ou seja é possível que as instruções dentro deste bloco possam ter várias instruções de salto incondicionais. Estes são saltos cuja instrução seguinte será sempre mesma. Por esta razão não são considerados como saídas do bloco; para isso existem apenas os

saltos condicionais (*branchs*). Neste caso, a próxima instrução no ciclo de execução vai depender da condição testada no *branch* e portanto são considerados uma saída ao bloco.

2.2.2 Definição de Megabloco

O Megabloco [10][1] é uma sequência de instruções que se repete um número significativo de vezes no *trace* do código em execução. Estas instruções geralmente são geradas por um *loop* no programa. No entanto a repetição inerente a cada ciclo no código não é a única forma acontecer um padrão repetido no *trace* do programa. Consoante vários testes dentro destes podem também existir vários testes em saltos condicionais, produzindo padrões diferentes durante a execução. Cada padrão pode dar origem a um Megabloco. Cada parte desse padrão são chamados de blocos básicos. Em cada bloco básico é possível sair da execução padrão, ou seja o Megabloco é um padrão com apenas um entrada mas com várias saídas. Na figura 2.4 temos uma instrução inicial que é o início do Megabloco e do primeiro bloco básico, até ao primeiro *branch* na linha 4. Um bloco básico continua a partir dessa linha que termina noutro *branch* na linha 7 (e, neste caso, acaba também o Megabloco).

```
1. 0x180 bslli r3, r4, 1026
2. 0x184 lw r3, r5, r3
3. 0x188 cmp r18, r3, r7
4. 0x18C bgeid r18, 12
5. 0x190 addik r4, r4, 1
6. 0x198 rsubk r18, r4, r6
7. 0x19C bnei r18, -28
```

Figura 2.4: Código do Microblaze em *assembly* a representar um Megabloco (fonte:[10])

Por estas razões testar simplesmente por uma instrução de *branch* não é eficaz em detetar Megablock, uma vez que a formação destes pode acontecer de várias formas, desde funções recursivas, ao próprio *loop* ou até a *loops* desenrolados.

2.2.3 Algoritmo de deteção

Detetar um Megabloco é um problema de detetar padrões. Estes padrões podem ser formados por vários tipos de unidades desde blocos básicos, fragmentos ou até apenas instruções. Na figura 2.5 temos vários exemplos de Megablocos. Por exemplo o Megabloco z existe pela repetição do fragmento B, que em si têm dois blocos básico delimitados pela condição A<3.

Detetando cada um desses blocos básicos é possível construir uma sequência de repetição dessas blocos por exemplo, xxx, sendo x cada um dos blocos básicos. A repetição de um padrão, duas vezes consecutivas é chamado de *square*. No caso mostrado existe um *square* do padrão com o bloco básico x. Apesar de objetivo final ser detetar padrões com várias iterações, detetando um

square a probabilidade de mais iterações desse padrão se seguirem é elevada. Por esta razão após a deteção de um *square* considera-se que foi detetado um Megabloco. Esta forma de deteção foi pensada para mapear os Megablocos em tempo real.

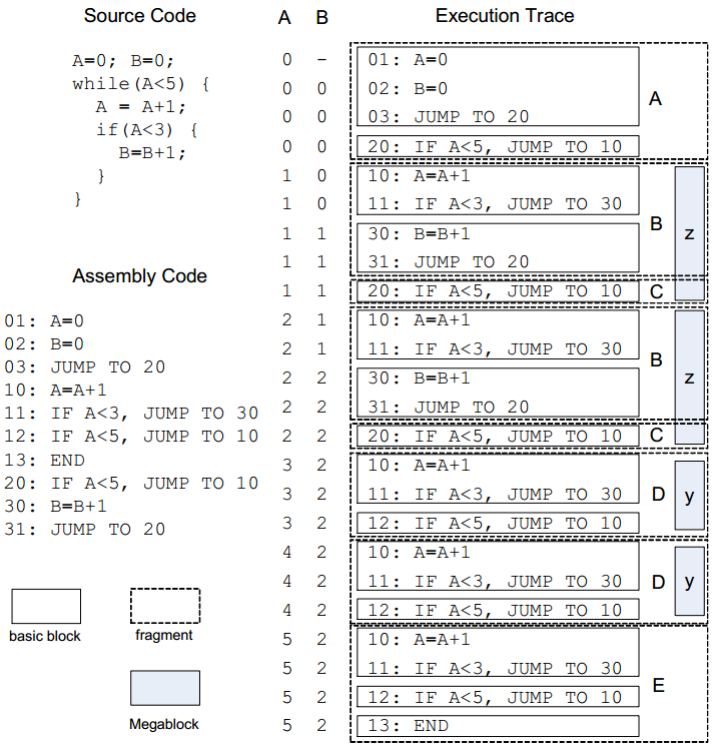

Figura 2.5: Programa executado com blocos básicos, fragmentos e os Megablocos formados por estes.(fonte:[10])

Considerando que podem existir vários padrões com o mesmo bloco básico, após a deteção de um *square* o padrão atual é guardado numa memória. Quando as repetições terminam, o algoritmo espera pelo começo de outro *square* e guarda o anterior. É possível encontrar padrões encadeados por exemplo,.. *aaaaaa*, encontra-se um padrão de tamanho 1 o *a*, tamanho 2 *aa*, ou até de tamanho 3 *aaa*. Por esta razão é necessário decidir qual dos padrões guardar: ou um padrão maior ou mais pequeno. A decisão é feita consoante se pretenda um *loop* interno ou *loop* desenrolados como prioridade. Quando a prioridade definida for para padrões maiores significa que se está espera de *loops* desenrolados. Por exemplo, *baabaa*, resulta no Megabloco *baa*. Se existir um padrao ,por exemplo *aaaa*, o padrão a detetar deve ser sempre apenas a.

Após a deteção e atualização das informações de cada Megabloco, cada um é guardado em memória. Esta informação dá a devida importância a cada Megabloco encontrado, comparando o número de iterações que cada um tem, sendo depois possível escolher quais passar para aceleração em hardware.

Capítulo 3

Conceção e Implementação

3.1 Requisitos

Considerando a implementação de uma tese anterior [1], onde um componente de acelerador foi criado para um sistema com um Microblaze, havia que estabelecer e guardar uma estrutura de dados chamada Megabloco. Essa estrutura existe para determinar os chamados *hot spots* de execução de sistemas embebidos. A aceleração total de uma aplicação pelo uso de um acelerador de hardware é dada pela quantidade do ciclo de execução que é movida para esse acelerador. A maneira eficiente de acelerar a execução é movendo grandes ou pequenas amostras de código que sejam as partes mais executadas de forma contínua pelo sistema.

A maneira de mudar a rotina para o acelerador tem uma sobrecarga inerente e por isso, pequenos grupos de instruções são ineficientes devido ao atraso inerente. Na implementação de um detetor Megablocos, deve haver determinados parâmetros ajustáveis: tamanho máximo do padrão, tipo de unidade de padrão e desenrolamento de *loops* internos. Estes parâmetros são necessários para a tese anterior, onde foram usados para determinar qual acelerador seria usado para otimizar o ciclo de execução. Para tal é necessário determinar algumas estatísticas além de detetar cada Megabloco individualmente: é preciso saber os números de iterações de cada padrão, tanto máximo como total. Estas servem para detetar a situação em que um Megabloco é executado intermitentemente, (i.e. acontece muitas vezes, mas com poucas repetições de cada vez) e através disso conhecer os casos que pode ou não sere vantajoso a acelerar.

3.2 Opções do projeto

De forma a construir um sistema capaz de substituir a solução existente em software é necessário que o sistema consiga encontrar os mesmos Megablocos utilizando poucos recursos. Desta forma e tendo em consideração como um Megabloco é formado, chegou-se à decisão de utilizar o mínimo de área possível.

Observando o conteúdo de Megablocos obtidos pelo programa de extração, ou seja as suas instruções e respetivos endereços, repara-se que estes são constituídos por grandes números de

```
1  0x00000228 lwi r5, r11, 0     //Inicio do Bloco Básico
2  0x0000022C lwi r3, r12, 0
3  0x00000230 addik r10, r10, 1
4  0x00000234 addik r12, r12, 4
5  0x00000238 rsubk r3, r5, r3
6  0x0000023C bsrli r4, r3, 31
7  0x00000240 rsubk r6, r3, r0
8  0x00000244 pcmpeq r5, r4, r0
9  0x00000248 mul r3, r3, r5
10 0x0000024C mul r4, r4, r6
11 0x00000250 addik r11, r11, 4
12 0x00000254 or r4, r4, r3
13 0x00000258 addk r7, r7, r4
14 0x0000025C xori r18, r10, 16
15 0x00000260 bneid r18, -56     // intrução de branch com um delay slot
16 0x00000264 swi r7, r8, 0      // FIM do Bloco Básico
```

Figura 3.1: Examplo do benchmark Motion_estimation-O2 Megablock:Endereço inicial 0x00000228

instruções . O Megabloco da figura 3.1 é composto por 1 bloco básico de 16 instruções, sendo que cada instrução tem atribuída um endereço de 32 bits. Para guardar a sua informação seriam necessários 32x16 =512 bits. Sabendo que existem Megablocos com vários blocos básicos, cada um com número diferente de instruções, a quantidade de memória necessária para guardar todos os tipos de Megablocos pode crescer muito.

No entanto, se em vez de serem guardados todos os endereços que compõe o Megabloco forem guardados os blocos básicos correspondentes consegue-se uma redução da área. Isto é possível porque existe um ficheiro executável do programa a ser corrido pelo sistema na memória da FPGA. Este ficheiro contem a informação necessária para que o código corra e outras funções tal como os endereços e instruções do código a executar. Com os endereços de iniciais e finais de cada bloco básico que faz parte do Megabloco é possível reconstruir o Megabloco completo. No caso do bloco básico da figura 3.1, este podia ser representado em 64 bits, 32 do início e 32 do fim, ficando então:0x00000228;0x00000264; Desta forma em vez de serem necessários 512 bits para o obter informações sobre o rastro de execução e sobre cada Megabloco, apenas seriam precisos 64 bits (menos 87.5% de recursos). No exemplo em anexo B.1, encontra-se um Megabloco constituído por nove blocos básicos com um total de 155 instruções. Neste iriam ser necessários 155*32 = 4960 bits ao contrário de 9*64=576 bits, o que daria um aumento de 88.3% em termos de bits necessários para guardar o padrão.

A memória com a informação de cada Megabloco contem o número de iterações máximo e total. Por exemplo, sendo *a* um Megabloco e se durante o *trace* de execução aparecer, *aaaxaa*, então o número máximo seria 3(*aaa*) e o total 5. Esta informação só pode ser atualizada se cada Megabloco for identificável. Assim escolheu-se usar como como referência de cada Megabloco o seu bloco básico com o endereço mais baixo.

3.3 Sistema final

O problema em se encontrar Megablocos é solucionado por um detetor de padrões. Para tal, é necessário primeiro detetar os blocos básicos do *trace*, a primeira fase do algoritmo de deteção. Estes blocos estão dependentes das instruções de *branch* do Microblaze ou qualquer outro *soft processor* instalado (módulo *BB_detect*, figura 3.2).

Aquando da deteção de um bloco básico, este é passado para o próximo módulo onde se verifica se um ou vários padrões existem e quais os seus tamanhos; isto equivale a encontrar um *square* na sequência de blocos básicos (segunda parte do algoritmo). Após deteção existe uma máquina de estados finita a contar o número de iterações de cada padrão. Deixando este de ocorrer (aparecer um bloco básico diferente dos existentes no padrão) e havendo iterações suficientes, procede-se à escrita na memória (módulo de *Megablock_pattern_detect*).

Na memória, existe a informação sobre todos os Megablocos detetados, qual o seu número de iterações máximo e o total (última parte do algoritmo deteção de Megablocos). Além disso, a memória tem a informação necessária para formar os Megablocos com os blocos básicos. A memória, tal como o os outros módulos, foi parametrizada em termos de tamanhos de padrões a detetar e guardar (módulo *Trace buffer*).

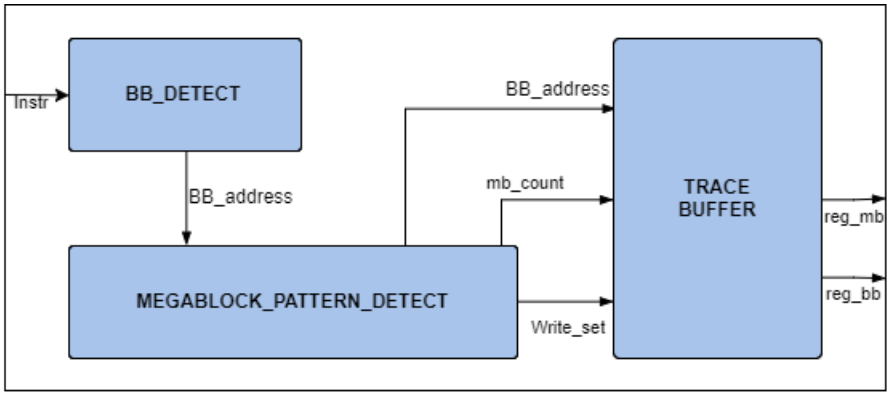

Figura 3.2: Módulo de topo: Megabloco extrator

Os parâmetros do projeto são:

- ADDRESS_SIZE -> Número de bits por endereço;
- INSTR -> Número de bits de cada instrução (trata-se de um processador RISC em que todas as instruções têm o mesmo tamanho);
- COUNT_SIZE -> Número de bits dos contadores de iterações;
- MAX_MB -> Tamanho máximo de Megablocos para detetar (em números de blocos básicos);
- TOTAL_MBs -> Número máximo de Megablocos distintos para guardar em memória;
- MEMORY -> Número máximo de blocos básicos para guardar em memória;

3.3.1 Basic Block Detector

É este módulo que produz os dados como o primeiro e o último endereço de cada bloco básico (tabela 3.1). Além disso, o sinal mais importante a ser propagado é o que escolhe o primeiro endereço de cada bloco básico e ao mesmo tempo dá a indicação que as cadeias seguintes do *Megablock pattern_detect* devem ser atualizadas, *delay_en*.

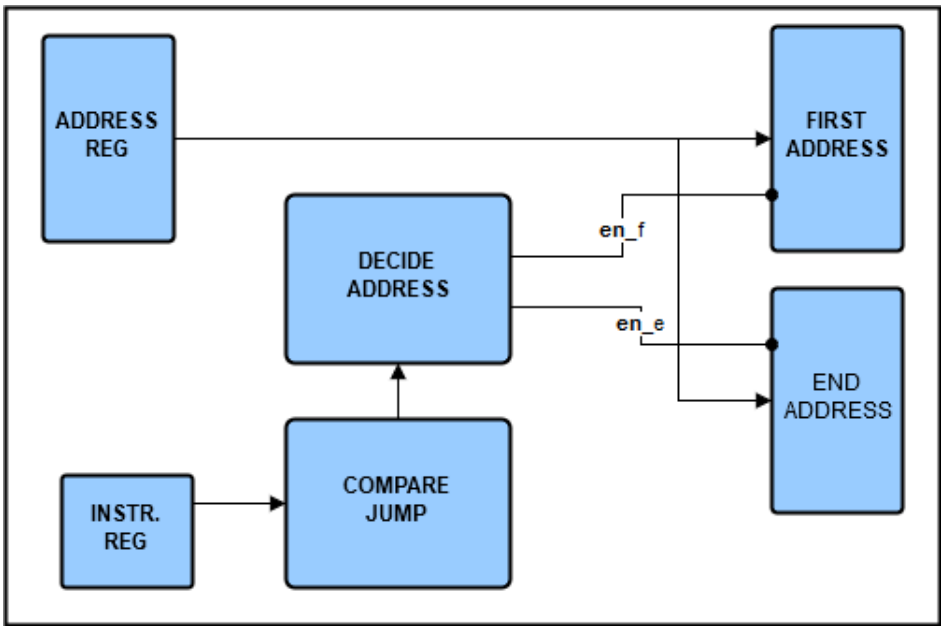

Figura 3.3: Módulo Basic block detector

Tabela 3.1: Interface Módulo BB_detect, Clock e Reset excluídos

Pin	Tipo	bits	descrição
enable	input	1	Sinal enable, indicando instrução válida e executada para tratar
instr	input	6	Sinal com os bits das instruções necessárias para detetar o bloco básico
adr	input	32	Sinal com os endereço relativo a cada instrução na execução do trace
start	input	1	Sinaliza a primeira instrução do primeiro bloco básico
delay_en	output	1	Indica que um bloco básico atual está pronto para ser processado
bb_init_adr	output	32	Endereço inicial do bloco básico para ser propagado no detetor de padrões
bb_end_adr	output	32	Endereço final do bloco básico para ser propagado no detetor de padrões

Assim sendo, este módulo funciona com a condição de conhecer a priori as instruções de *branch* e que *delay slots* estas podem ter. Para o caso das instruções do Microblaze [11] só podemos ter zero ou um *delay slot*. Um *delay slot* é um espaço para instrução a seguir ao *branch* ser executada sempre, independentemente do resultado da comparação no *branch*. Podem existir no entanto outras arquiteturas com 2 *delay slots*, como por exemplo a SHARC.

Os sinais recebidos de *start* e *enable* são usados para saber o início do primeiro bloco básico e se o existe uma instrução para ser processada, respetivamente. Com esses sinais e os dados recebidos de cada instrução e endereço é possível encontrar os bloco básico (figura 3.3).

O modo de funcionamento centra-se no facto de cada bloco básico acabar com uma instrução de *branch*; em caso de existir um *delay slot* apenas a instrução seguinte termina o bloco. Com isto, cada instrução válida e endereço correspondente sinalizado pelo sinal de *enable* são comparados aos registos pré-estabelecidos com a informação necessária para decidir se esta é uma instrução de *branch* ou não. Simultaneamente é verificada a existência de *delay slot* na instrução caso esta seja um *branch*.

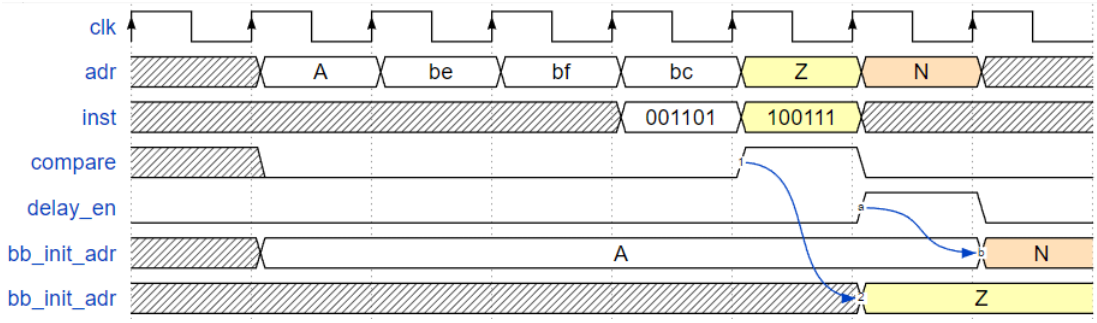

Figura 3.4: Exemplo de um caso deteção do início e fim de um bloco básico sem *delay slot*

Com estes sinais é possível registar o endereço inicial do bloco básico dependendo de se um *delay slot* foi encontrado ou não, e ao mesmo tempo registar o endereço final. Estes dados são de seguida passados ao Megablock pattern_detect, onde o sinal de *delay_en* ajuda a propagar pela cadeia do módulo Square Detector. Por exemplo na figura 3.4 mostra-se a situação em que se encontra a combinação de bits numa instrução que indica que um *branch* foi detetado. O sinal que indica a atualização, *delay_en*, do próximo de bloco básico é também o que indica a deslocação dos blocos na cadeia do próximo módulo.

3.3.2 Módulo Megablock Pattern_Detect

Neste módulo, ocorre a maior parte da deteção dos Megablocos e das suas iterações. Esta deteção é feita pelo módulo Square Detector. Este passa a informação de tamanho de cada megabloco ao Encoder e por fim este passa informação mais compacta sobre o tamanho para as máquinas de estado dentro do módulo Control.

Como se pode ver na figura 3.5, existe o sinal *pattern_size*. Este passa do Square detector para o Encoder e do *pattern* do Encoder para Control, sofrendo certas alterações de forma a poder ser utilizado pelas FSMs. O Square detetor é atualizado todas as vezes que o BB_Detect ativa o sinal de *en_bb_address* e propaga a cadeia de blocos básicos, enquanto tenta descobrir um *square* na cadeia.

A indicação de padrão detetado é passada ao módulo Encoder através de *pattern_size*, para este decidir no caso de dois padrões detetados de tamanhos diferentes qual será o padrão escolhido. Esta escolha é controlada pelo sinal, *Big_pattern*: se estiver ativo é escolhido o padrão de maior tamanho, se não houver um padrão atualmente detetado; caso contrário o padrão atual tem prioridade. Este caso pode acontecer porque na sequência de alguns padrões de blocos básicos como por exemplo *aaa*, seria possível escolher entre*a, aa, aaa*.

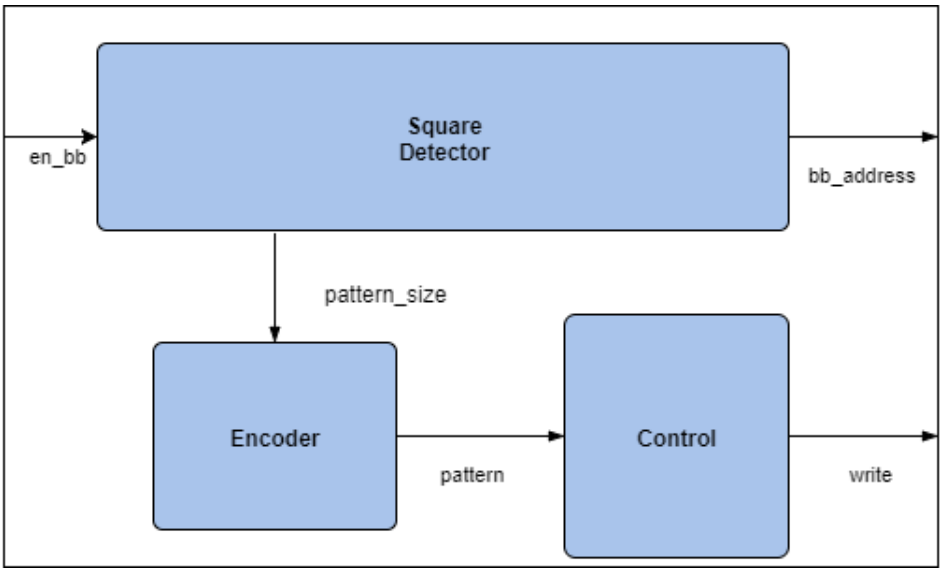

Figura 3.5: Módulo Megablock pattern_detect

O padrão depois de codificado é usado nas máquinas de estado em Control para verificar cada Megabloco. A mudança entre padrões ou facto deste deixar de ocorrer implica uma alteração de estados ou de valores utilizados pelas várias máquinas de estados.

3.3.2.1 Módulo Square detetor

O módulo em questão tem por objetivo detetar um padrão chamado *squares*, a repetição duas vezes de um conjunto de blocos básicos, que indica a possibilidade de um Megablocos encontrado. A partir de o momento em que um square existe no *trace* de execução, a probabilidade de se tornar em Megabloco é elevada e portanto considera-se o padrão detetado um Megabloco.

Devido a esta possibilidade, este módulo consegue detetar que certos *squares* não são Megablocos. Por isso, existe o módulo Control que elimina esses falsas deteções. No fim da iterações de cada padrão, é decidido se o padrão é um Megabloco, ou seja, se passou o limite previamente estipulado. Este limite para o projeto foi de 5 iterações consecutivas.

Tabela 3.2: Interface Módulo Square_detector, Clock e Reset excluídos

Pin	Tipo	bits	descrição
bb_address	output	64	Endereços inicial e final do blocos básicos utilizados na deteção de padrões
bb_address_out	output	64	Endereços inicial e final do blocos básicos que serão gravados na memória
en_bb_address	input	1	Sinal que propaga sequencia a deteção de padrões
pattern_size	input	16	Vetor em que cada sinal indica se o padrão do tamanho x foi encontrado

Este módulo é essencialmente composto por uma cadeia de registos, contendo cada um o endereço final e inicial de um bloco básico. A cadeia propaga dependendo do sinal *en_bb_address* que recebe e envia cada informação do bloco básico para o nível seguinte até ao ultimo nível, como se pode verificar na figura 3.6. É passado entre níveis tanto o início do bloco básico como

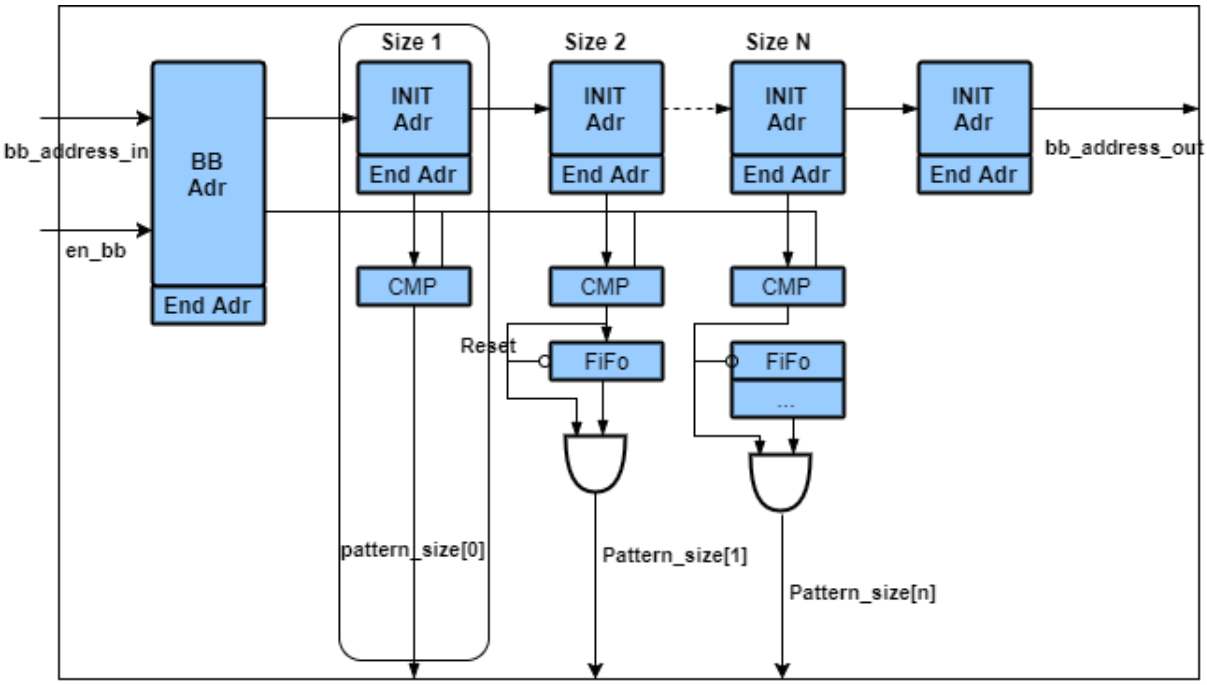

Figura 3.6: Módulo Square_detector

também o fim dele. A saída deste módulo com os endereços é usada para guardar as informações de cada Megabloco (tabela 3.2).

O primeiro elemento da cadeia é usado como comparador com todos níveis da mesma; se algum elemento da cadeia for igual, o resultado será guardado na FIFO de tamanho x-1 igual ao tamanho do padrão, exceto no primeiro nível onde existe apenas a comparação com registo inicial. Esta FIFO serve para guardar a indicação de blocos básicos intermédios de padrões que foram encontrados. Após se encontrar o correspondente bloco básico no nível do tamanho do padrão (figura 3.7) sem haver falhas de comparação pelo meio, este vai sinalizar que foi encontrado o padrão de tamanho x.

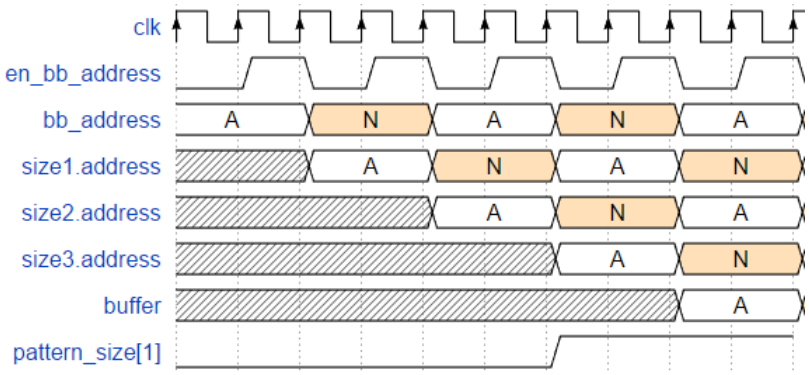

Figura 3.7: Diagrama temporal: Módulo Square_detector

A operação lógica AND serve para verificar se o padrão ocorreu. A FIFO fica a zero quando o

elemento de cada nível não for igual ao início. Se o tal padrão acontecer a FIFO enche e o último elemento desta iguala o sinal de comparação e é sinalizado que o padrão foi encontrado.

Como se pode reparar no módulo de Square detetor, existe uma cadeia relativamente grande de registos de 64 bits (endereço inicial e final de cada bloco básico) cada e comparações com esses registos. Nessa cadeia encontrou-se o caminho crítico do circuito. Inicialmente estas comparações e o módulo Encoder encontravam-se ligados de forma combinacional, sem registos entre os dois ou nalgum deles. Por isso, era normal encontrar um tempo de propagação elevado neste caminho. De forma a corrigir esta falha, procedeu-se da seguinte forma: acrescentou-se um registo a mais para os sinais no Square detetor e ao mesmo tempo um registo para o sinal *pattern_size* no módulo de Encoder. Desta forma cortou-se o caminho mais longo entre os dois módulos e mantiveram-se as propriedades do sistema todo, sem mudar nenhuma outra questão de sincronização.

3.3.2.2 Módulo Encoder

O módulo relativamente mais simples do sistema torna os sinais recebidos pelo Square_detector (tabela 3.3) em algo mais fácil de analisar. Se o Square_detector detetar dois ou mais Megablocos com tamanhos diferentes, cabe a este módulo decidir qual escolher.

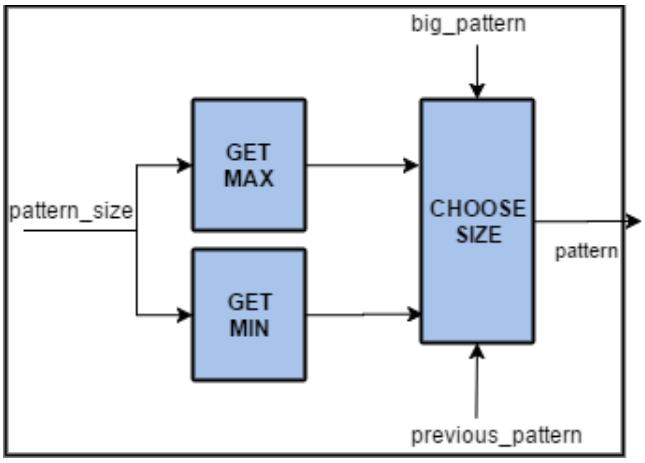

Figura 3.8: Módulo encoder

Tabela 3.3: Interface Módulo Encoder, Clock e Reset excluídos

Pin	Tipo	Bits	descrição
en_bb_address	input	1	Utilizado para atualizar o padrão atual
big_pattern	input	1	Escolhe entre os tamanhos de padrões atuais caso existem dois ou mais ao mesmo tempo.
pattern_size	input	16	Indica quais dos tamanhos de padrões foram encontrados
previous_pattern	input	4	Padrão encontrado anterioremente,
pattern	output	4	Atualiza o padrão atual

Essa escolha depende do sinal *big_pattern* (figura 3.8). Em caso ativo o módulo irá escolher padrões maiores sempre que não houver sido detetado anteriormente um padrão; se assim for e o

mesmo padrão continuar a ser detetado, este não é mudado. No caso do sinal se encontrar inativo o módulo irá escolher sempre o padrão mais pequeno.

Listing 3.1: Código em Verilog da escolha entre padrões

```
always @(*) begin
            if(big_pattern && (pattern_size!=0)) begin
             // verifica se o padrões maiores são primários
                if( (previous_pattern > counter_min) &&
                 (previous_pattern == counter_max ) ) begin
                 // verifica se o padrão em memória ainda está ativo
                        pattern = counter_max   ;
                end
                else    begin
                        pattern = counter_min           ;
                end
            end

            else if(pattern_size!=0) begin
             // se não, então escolha o padrão menor
                pattern = counter_min    ;
            end
            else begin
                pattern = 0             ;
            end
        end
```

3.3.2.3 Módulo Control

As máquinas de estados são o cérebro deste sistema. Existem quatro FSM a controlar o sistema, sendo que estas trabalham aos pares. Cada par tem uma FSM que controla a escrita do Megabloco, Pattern_control, e outra a contagem de iterações, Counter, tabela 3.4. Os pares trocam de ativo para inativo em momentos separados de forma a não existirem iterações de um Megabloco por contar.

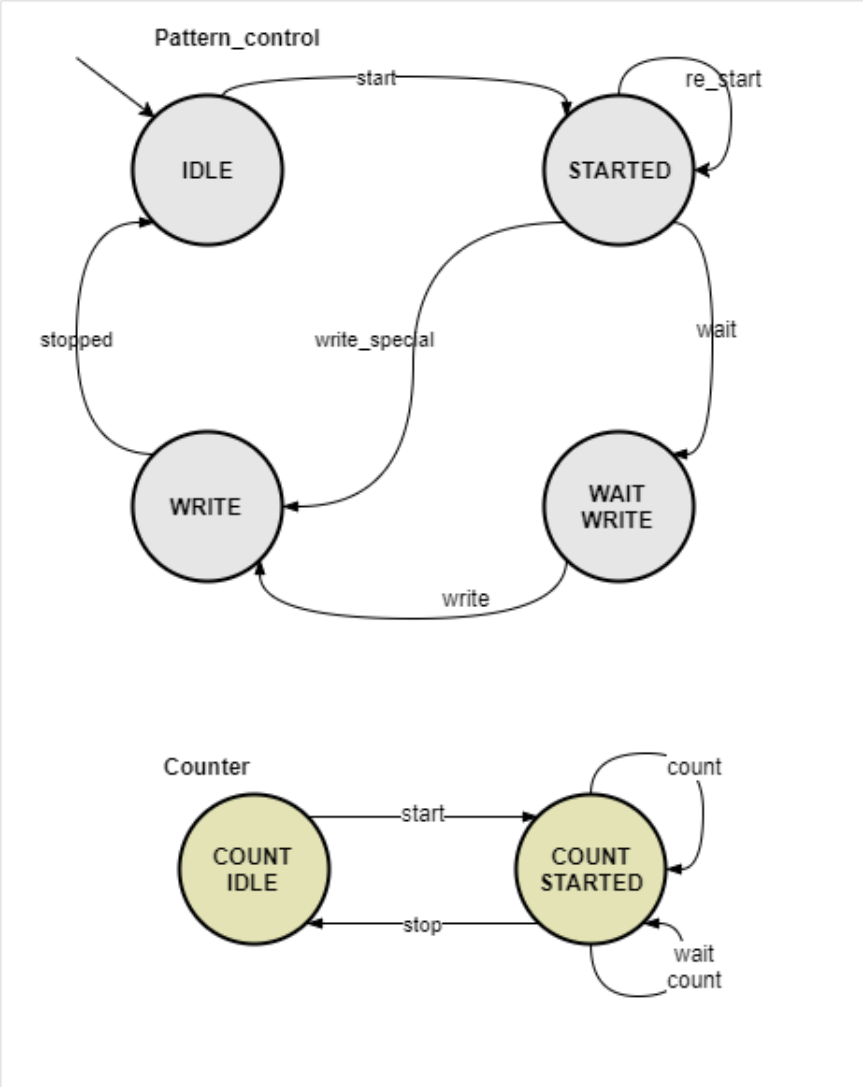

Figura 3.9: Pattern_control: máquina de estados que sinaliza quando escrever na memória, Counter: contador do número de repetições.

Tabela 3.4: Interface Módulo Control, Clock e Reset excluídos

Pin	Tipo	bits	descrição
pattern	input	16	Indica o tamanho do padrão atual, se este existir
en_bb_address	input	1	Sinal utilizado como enable da máquina de estados, ou seja esta só muda de estado com um impulso deste sinal
enable	input	1	Indica quando a máquina de estados deve parar se este sinal estiver inativo
write_set	output	1	Indica que é necessário escrever na memória
mb_size	output	16	Tamanho do Megabloco que irá ser colocado em memória
pattern_now	output	16	Tamanho do Megabloco que foi detetado mas ainda vai ser escrito em memória
index_var	output	16	Indica o variação de índice de memória, muda conforme o bloco básico do padrão a ser escrito
end_fsm	output	1	Sinal usado para simular en_bb_address caso ainda exista um Megabloco por escrever em memória e enable esteja inativo.
done	output	1	Sinal que implica o fim da deteção de Megablocos.
count_hold	output	16	Indica o número de iterações do padrão atual que irá ser colocado em memória

Tabela 3.5: Transições das Máquinas de estado FSM e Counter

Transição	descrição
start	Quando pattern !=0 e a FSM complementar não estiver ativa. Esta torna-se ativa guardando o padrão atual
re_start	Após um valor de pattern diferente do guardado e não houver iterações suficientes, mantém se no mesmo estado com um novo padrão
wait	Após um valor de pattern diferente do guardado, este muda para o estado WRITE se o número de iteraçoes for maior que um limite.
write	Quando for momento da escrita, contado por uma variavél auxiliar, este muda de estado para WRITE
write_special	Após um valor de pattern diferente do guardado mas este for igual ao tamanho máximo de Megablocos Este muda para o estado WRITE se o número de iteraçoes for maior que um limite.
stopped	Após terminar a escrita retorna a IDLE

count_start	Quando pattern !=0 e a FSM complementar não estiver em ativo. Esta torna-se ativa guardando o padrão atual e mudando de estado para COUNT_STARTED
count	Após contar certos ciclos dependendo do padrão incrementa o contador das iterações
wait_count	Após um valor de pattern differente do guardado e não houver iterações suficientes mas o novo padrão ser diferente de 0, mantém se no mesmo estado com um novo padrão
stop	Após um valor de pattern differente do guardado e este for zero volta a COUNT_IDLE

O conceito fundamental que faz este módulo funcionar é a mudança de valor de *pattern* (tabela 3.5). Este sinal é o que têm mais impacto nestas máquinas de estado. Após ser diferente de zero, o primeiro par Pattern_control e Counter começam a trabalhar. Por serem dois pares semelhantes de FSM existe um circuito combinacional a controlar as saídas do módulo, pois estas são dependentes de valores dos dois pares de FSM. Tanto um par como o outro indicam qual o número de iterações do Megabloco atual e quando este deve ser escrito em memória.

Cada par de FSM começa a contar quando o outro não estiver a trabalhar, ou seja quando Pattern_control estiver em modo de WAIT_WRITE ou WRITE (figura 3.9) o outro par pode começar. Este dualidade é necessária de forma a não se perderem iterações de padrões intermitentes ou padrões que se formem enquanto está ser escrito o mais recente Megabloco,3.10:

3.3.3 Trace Buffer

Este módulo tem como objetivo guardar as informações necessárias para a reconstrução do Megabloco, bem como o seu tamanho e número de iterações. As estatísticas sobre esses Megablocos são importantes para decidir se cada Megabloco encontrado deve ou não ser acelerado em hardware, figura 3.11.

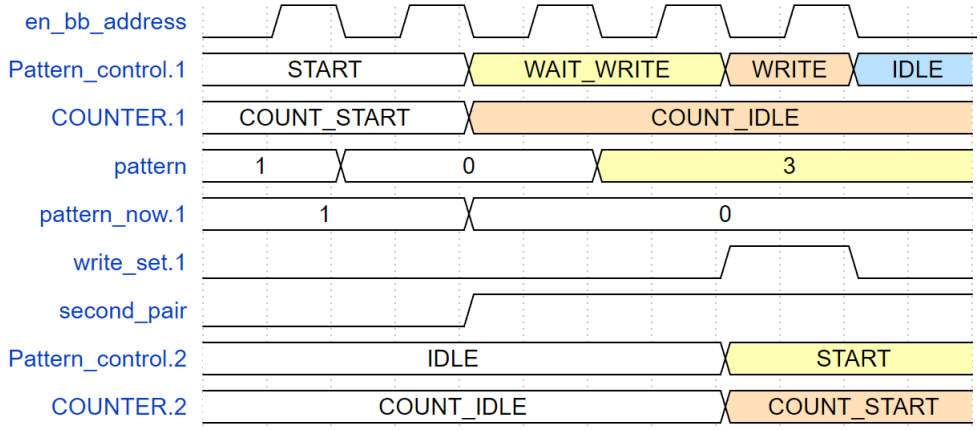

Figura 3.10: FSM: máquina de estado que sinaliza quando escrever na memória, Counter: contador do número de iterações.

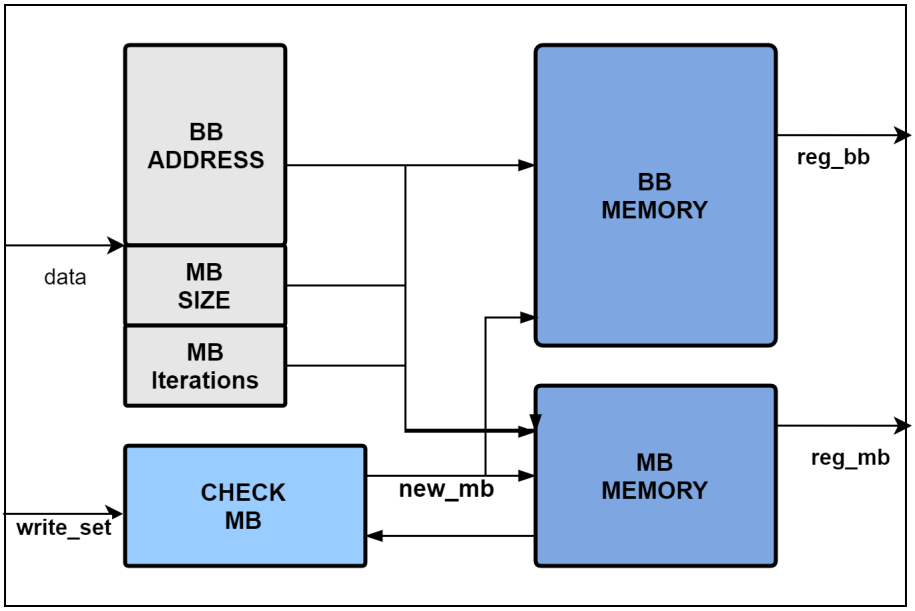

Figura 3.11: Diagrama de blocos do módulo Trace buffer

Os sinais importantes neste módulo são *bb_adr_inst* e *write_set* (tabela 3.6). Após se iniciar o registo de um novo Megabloco, é guardado o seu tamanho. No fim da escrita do Megabloco atual, o tamanho do Megabloco guardado é utilizado para aumentar o índice da memória de blocos básicos, BB_memory. São escritos nesta memória os endereços de Blocos Básicos num índice novo a cada mudança destes enquanto *write_set* ativo. A atualização no fim da escrita só acontece se não houver um Megabloco na memória.

O bloco *MB_Memory* guarda as estatísticas importantes de cada Megabloco. Cada posição de memória contém o endereço mais baixo de cada Megabloco, que é usado para identificar cada a um. Além disso tem o tamanho do padrão, o número de iterações total e máximo e o número de ocorrências do Megabloco. Nesta bloco verifica-se se existe em memória um padrão igual ao atual; em tal caso, o índice desta memória não incrementa e acontece a atualização do número de

Tabela 3.6: Interface Módulo Trace_buffer, Clock e Reset excluidos

Pin	Tipo	bits	descrição
bb_adr_inst	input	64	Endereços inicial e final do bloco básico que serão gravados na memória
write_set	input	1	Sinaliza escrita dos blocos básicos que compõem o Megabloco atual
mb_size	input	16	Tamanho do Megabloco atual
mb_count	input	16	Indica o número de iterações do Megabloco atual
index_for_bb	input	8	Índice usado para retirar os dados de reg_bb
index_for_mb	input	8	Índice usado para retirar os dados de reg_mb
index_var	input	8	Indica a posição do bloco básico a ser escrito
trace_mb	output	78	Sinais com os valores da memória com as estatísticas de cada Megabloco
trace_bb	output	64	Sinais com os valores da memória com os blocos básicos pertencentes a cada padrão

ocorrências do padrão tal como do número de iterações máximo e total.

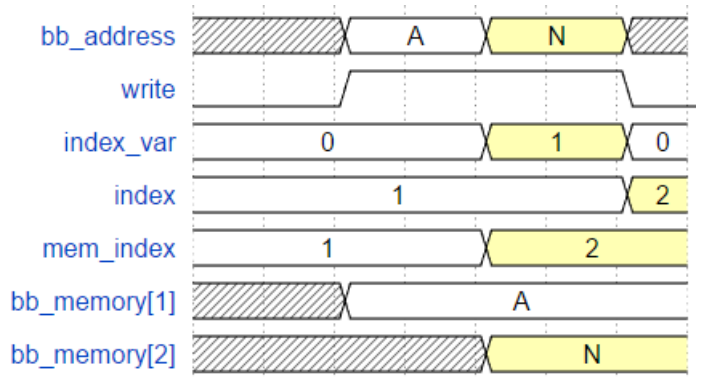

Figura 3.12: Diagrama temporal: Módulo MB memory

Após a deteção de Megablocos, é possível extrair estes dados com sinais de índice das memórias, começando por primeiro determinar o último índice utilizado em MB memory. A partir disso pode retirar-se o tamanho do último Megabloco e saber quantos dos elementos retirar dos últimos blocos básicos em BB memory (figura 3.12).

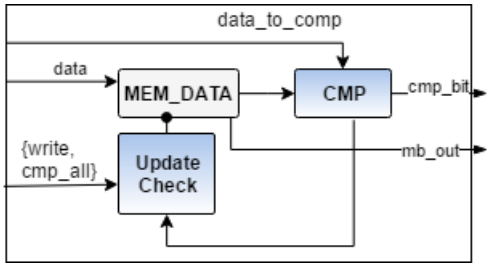

Figura 3.13: Componentes de MB memory.

Na figura 3.13 encontra-se um sub-módulo que faz parte do bloco MB memory, sendo que este último é replicado várias vezes em conjunto com uma simples porta lógica OR. Os sub-módulos são comandados por índice que indica qual será deles o próximo a ser preenchido. O índice é atualizado após ser encontrado um novo Megabloco.

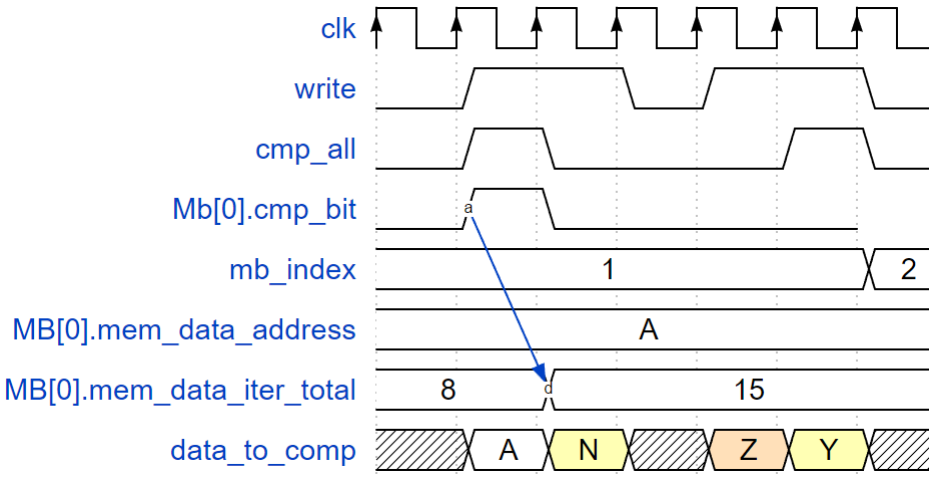

Figura 3.14: Bloco Mb Memory.

Cada componente de MB_memory, verifica se o bloco básico atual e que representa o Megabloco anteriormente é igual ao que tem em memória (figura 3.14). Se assim for atualiza os dados de iterações máximas e totais desse Megabloco. Ao mesmo tempo o*cmp_bit* ficaria ativo e então o *cmp_all*, este é simplesmente a operação lógica OR de todos os *cmp_bits*, estará também ativo e o próximo índice não irá ser atualizado, uma vez que foi entendido que não houve um novo Megabloco.

Um dos problemas com esta abordagem para as memórias tem por base os contadores de iterações. Estes, no caso de alguns *traces* grandes, podem fazer *overflow* e ao mesmo tempo existir outro Megabloco em memória que assim passa a ter mais iterações que o primeiro. Contudo a situação é rara e não foi tratada explicitamente.

Outro dos problemas deve-se à escolha feita para identificar Megablocos. Em casos particulares de alguns *traces* pode acontecer que existam dois Megablocos, relevantes tais que os seus endereços mais baixos sejam iguais. Assim quando acontecer a primeira deteção de um desses Megablocos, irá ter os seus valores de iterações corretos, mas na segunda, se o próximo Megabloco diferente tem um bloco básico dentro dele igual ao identificador do outro, os valores irão ficar incorretos para o primeiro.

Para o fim a que se destina a informação recolhida, considerou-se que também esta situação ocorrerá com pouca frequência, pelo que não foi tratada nesta versão de hardware.

Capítulo 4

Resultados

4.1 Bancada de teste e resultados

Em todos os sistemas de digitais será necessário fornecer um teste para verificar a sua funcionalidade e é para isso que a bancada de teste existe. Devido à existência prévia da aplicação de extração de Megablocos que este sistema replica em hardware, um objetivo interessante foi a a comparação em termos de Megablocos encontrados e número de iterações. A aplicação neste momento gera por simulação um trace com os endereços e as respetivas instruções em hexadecimal para cada benchmark testado. Com esse trace é possível testar o módulo em hardware e comparar os resultados dos dois.

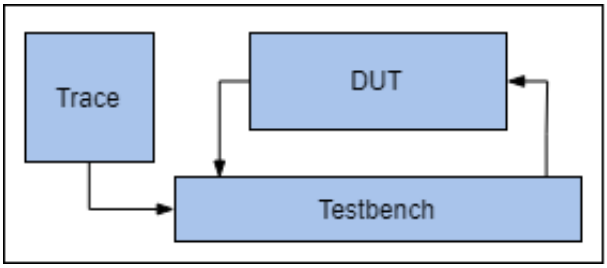

Figura 4.1: Testbench e sua consolidação com o DUT

A bancada de teste tem um objetivo principal que é enviar o *trace* e o resto dos sinais necessários ao funcionamento do módulo a testar. O *trace* em si tinha uma forma específica que precisou de ser modificada para ficar de acordo com os sinais de entrada do módulo tal como pode ser visto a seguir.

- 33:0x000000E8:0xBE030010

Por isso foi preciso decompor esta informação em algo relevante ao módulo. Neste caso transformar as instruções em hexadecimal para binário e enviar ao módulo. Para isso foram usadas certas funções apenas existentes em System Verilog e que trabalham sobre cadeias de carateres. Uma dessas funções é a que faz o primeiro tratamento de cada linha, ou seja cada instrução do trace e separa as instruções dos endereços.

Listing 4.1: Função: Encontrar endereço e instruções no trace

```
function string adr_inst (string str, int ab);
      automatic integer first  =0  ;
      automatic integer sec    =0 ;
      automatic integer d   =0;
      string  adr     ;
       for(int i=0; i<str.len(); i=i+1) begin
       // encontra primeiros delimitadores
       // que sinaliza onde se encontra o endereco
            if(str.getc(i)== 58 && (d== 0)) begin
               first   = i;
               d = 1;
            end
            if(str.getc(i) == 58 && (d==1)) begin
            // encontra onde comeca a instrucao
               sec = i;
            end
       end
         if(ab ==1) begin
         // verifica-se que enviar os dados do endereco ou instrucao
            adr = str.substr(first+3,sec-1);
            return adr;
         end
         else begin
         // envia os dados da instrucao
            adr = str.substr(sec+3, str.len()-1);
            return adr   ;
         end

   endfunction
```

Após este primeiro tratamento do *trace*, é possível começar a converter cada instrução e endereço de hexadecimal para binário. Com os primeiros destes dados tratados, a bancada de teste envia o sinal de *enable* e de *start* do bloco básico, de forma a que o módulo comece o seu trabalho. A cada flanco de relógio uma nova linha do *trace* é lida, tratada e enviada para o sistema.

4.1.1 Resultados funcionais

Os resultados encontrados dependem de vários parâmetros ou sinais, entre as quais o sinal *big_pattern*. Foi feita a comparação com os resultados obtidos em software.

Tabela 4.1: Megablocos encontrados pela aplicação e o sistema em hardware para alguns benchmarks

	count-O0	Fibonacci-O2	pop_cnt_8bit-O0	HammingDistance-O0	bubble_sort-O2
Software	0x000001C8 0x0000023C	0x00000194	0x000001CC 0x000001E0	0x000001DC	0x000001F4
Hardware	0x000001C8 0x0000023C	0x00000194	0x000001CC 0x000001E0 0x00000158	0x000001DC	0x000001F4
	Vecsum-O2	**Wave_vert-O2**	**Motion_estimation-O2**	**Quantize-O2**	**Median-O2**
Software	0x00000180	0x0000020C 0x000001F8 0x000001B4 0x000001A4	0x00000228	0x000001E4	0x00000390
Hardware	0x00000180	0x0000020C 0x000001F8 0x000001B4 0x000001A4	0x00000228	0x000001E4 0x00000158	0x00000390

Como se pode ver na tabela 4.1 os Megablocos encontrados são iguais aos que aplicação encontra com a exceção do Megabloco 0x0000001C8. Este faz parte da inicialização de vários dos benchmarks e tem um número de iterações consecutivas maior que o limite estabelecido. Por essa razão este é considerado um Megabloco pelo módulo em hardware.

Outras diferenças que se podem verificar na tabela 4.2 são nas iterações encontradas, que nem sempre são iguais. No entanto, apenas são diferentes por mais ou menos uma unidade. Isto acontece porque o sistema em hardware pode não reconhecer a ultima iteração de um megabloco, pois o bloco básico que termina o Megabloco atual pode não ser o último do padrão. Se tal acontecer, o módulo em hardware conta como mais uma iteração apesar de o padrão não estar completo nesta última. Um caso mais particular de diferenças em iterações deve-se à forma como este detetor de Megablocos decide quando incrementar iterações. Se o Megabloco "a"existir dentro de outro Megabloco, por exemplo "bac", então o detetor irá contar que este Megabloco é o "a"e que por isso incrementa o respetivo contador.

4.2 Recursos após Implementação

A utilidade de um sistema em hardware depende também dos seus recursos utilizados. Neste caso foram estudados os recursos pós-implementação para verificar que impacto as suas diferenças tinham nos parâmetros. As opções utilizadas no Vivado para produzir estes resultados foram escolhidas de forma a melhorar os tempos encontrados independentemente da área utilizada.

Como mostra a figura 4.3 o parâmetro com mais impacto em termos de recursos é o MAX_MB, que indica o tamanho máximo para um Megabloco contado em número de blocos básicos. O tamanho de um Megabloco para o sistema em questão modifica a cadeia dentro de o Square Detector e esta, sendo composta por registos de 64 bits em cada nível, mais uma FIFO a aumentar com o tamanho da cadeia e ainda alguns comparadores, tem um impacto significativo no número de FF e de LUTs.

Tabela 4.2: Resultados em número de iterações de Megablocos para alguns benchmarks

Benchmark	Megabloco	Hardware Total	Software Total	Hardware Max	Software Max
count-O0	0x000001C8	118	12	97	6
	0x0000023C	97	98	97	98
Fibonacci-O2	0x00000194	16630	16626	9995	9996
HamingDistance-O0	0x000001DC	3050	3000	31	30
	0x00000158	7	0	7	0
pop_cnt_8_bit-O0	0x000001CC	1022	1022	1022	1022
	0x000001E0	1042	12	1022	6
	0x00000158	7	0	7	0
Quantize-O2	0x000001E4	3152	3152	197	197
	0x00000158	7	0	7	0
Vecsum-O2	0x00000180	2046	2045	2046	2045
Wave_vert-O2	0x0000020C	44	10	38	5
	0x000001F8	38	37	38	37
	0x000001B4	51	10	38	5
	0x000001A4	38	38	38	38
Motion_estimation-O2	0x00000228	57344	53248	14	13
bubble_sort-O2	0x000001F4	3870	3840	61	60
Median-O2	0x00000390	997	997	997	997

Tabela 4.3: Relatório de recursos pós-implementação na placa Zedboard com Zynq-7020. Parâmetros Normal: ADDRESS_SIZE =32, INST_SIZE =7, COUNT_SIZE = 16, MAX_MB = 32, TOTAL_MB =16, MEMORY= 32

Parâmetros	Recurso	Utilizados	Disponiveis	Util %
Normal	**Lut**	1034	53200	1.94
	FF	1453	106400	1.37
MAX_MB 32	**Lut**	1412	53200	2.65
	FF	2375	106400	2.23
MEMORY 64	**Lut**	1029	53200	1.93
	FF	1454	106400	1.37
MEMORY 128	**Lut**	1040	53200	1.95
	FF	1466	106400	1.38
TOTAL_MB 32	**Lut**	1023	53200	1.92
	FF	1453	106400	1.37

4.2.1 Tempos

Dependendo dos parâmetros os escolhidos, o sistema consegue atingir frequências de operação iguais à do Microblaze. Um parâmetro com impacto nestes cálculos é MAX_MB. Este muda a cadeia de flip-flops e LUTs usados pelo Square_detector. Numa versão inicial, não existia um registo para saída desse módulo, no entanto após vários testes de otimização por parte das ferramentas não era possível chegar à frequência esperada. Para tal acontecer, colocaram-se alguns registos adicionais no percurso de dados do sistema.

O local ideal para tal ser feito é entre a cadeia do Square detector e o Encoder. A solução foi acrescentar um registo a mais na cadeia de Square detector para retardar a saída dos dados e ao mesmo tempo registar os sinais *pattern_size*. O atraso inerente ao registo dos padrões pode ser anulado pelo *buffer* adicional. Esta solução só resulta porque as FSMs mudam de estado com diferenças no sinal *pattern_size*. Se essa diferença for atrasada, as saídas das FSMs estarão atrasadas também e por isso o *buffer* dos dados no Square Detector no fim é necessário para ficar sincronizado com os dados nas FSMs.

4.2.2 Estimativas de Potência

Os registos de potência encontrados foram obtidos por a função de Report Power em Vivado Design Suite usando um ficheiro de atividade obtido por simulação pós-implementação. O relatório de potência apresentou um nível de confiança elevado.Os resultados destas simulações foram conseguidos com os seguintes benchmarks: Median-O2, bubble_sort-, Quantize-O2 e Wave_vert-O2.

Tabela 4.4: Potência consumida pelo sistema: Parâmetros Normal: ADDRESS_SIZE =32, INST_SIZE =7, COUNT_SIZE = 16, MAX_MB = 32, TOTAL_MB =16, MEMORY= 32

Power	**Static**	**Dynamic**	Clocks	Signals	Logic
0.124	0.120	0.004	88%	9%	3%

Devido aos *traces* usados nas simulações, a tabela 4.4 mostra que a energia consumida é maioritariamente em modo estático. Por cada *trace* ser relativamente pequeno e apenas excitar até mais ou menos 12 bits do endereços, a potência dinâmica tem menos importância que a potência estática. No entanto, o sistema total da FPGA é o que mais contribui para esta discrepância entre os resultados para potência estática e dinâmica. Uma vez que a FPGA é constituída por muitos componentes, apesar de muitos deles não serem usados nesta implementação, cada um continua a ter correntes de fuga. Com uma utilização de aproximadamente 2% já seria de esperar níveis baixos de potência dinâmica em relação à potência estática.

Capítulo 5

Conclusões e Trabalho Futuro

Este trabalho correu conforme previsto em termos de resultados e trabalho realizado. Criou-se um sistema de extração de Megablocos. Com informações por este encontradas, pode mais tarde construir-se um sistema completo com o *soft-processor* e o acelerador. Comparando com a solução anterior em software, são encontrados pelo módulo hardware os mesmos Megablocos dentro dos limites impostos pelos recursos.

Em termos de iterações, o sistema encontra valores quase iguais aos encontrados pela aplicação; como o mais importante é existir um termo de comparação entre os valores de cada padrão encontrado, estas diferenças não são significativas a nível do processo global. Em relação aos recursos da placa este sistema gasta até 2% desses recursos.

De forma a aperfeiçoar este módulo podemos fazer algumas alterações ao mesmo; sendo que no que diz respeito ao sistema completo em FPGA, o módulo desenvolvido de *profiling* precisa de interface com a criação do acelerador. Para isso é necessário que exista um módulo capaz de retirar as informações do extrator em hardware e com estes dados e as instruções do programa em memória reconstruir cada Megabloco, para depois este ser enviado ao sistema de configuração e gestão dos aceleradores.

Além disso, um dos problemas encontrados nesta implementação é forma como se identifica cada Megabloco. Este problema pode ser facilmente reduzido se em vez do identificador de cada Megabloco ser o bloco básico com endereço mais pequeno, passar a ser criado por uma função de dispersão calculada para cada bloco básico do Megabloco.

Anexo A

Exemplo de um Megabloco complexo

```
1   0x000001CC lwi r3, r19, 24  // inicio do Megabloco
2   0x000001D0 swi r3, r19, 12
3   0x000001D4 swi r0, r19, 16
4   0x000001DC bri 56
5   0x00000214 lwi r4, r19, 20
6   0x00000218 lwi r3, r19, 4
7   0x0000021C cmp r18, r3, r4
8   0x00000220 blti r18, -64
9   0x000001E0 lwi r3, r19, 12
10  0x000001E4 andi r4, r3, 1
11  0x000001E8 lwi r3, r19, 16
12  0x000001EC addk r3, r3, r4
13  0x000001F0 swi r3, r19, 16
14  0x000001F4 lwi r4, r19, 12
15  0x000001F8 bsrli r3, r4, 31
16  0x000001FC addk r3, r3, r4
17  0x00000200 sra r3, r3
18  0x00000204 swi r3, r19, 12
19  0x00000208 lwi r3, r19, 20
20  0x0000020C addik r3, r3, 1
21  0x00000210 swi r3, r19, 20
22  0x00000214 lwi r4, r19, 20
23  0x00000218 lwi r3, r19, 4
24  0x0000021C cmp r18, r3, r4
25  0x00000220 blti r18, -64
26  0x000001E0 lwi r3, r19, 12
27  0x000001E4 andi r4, r3, 1
28  0x000001E8 lwi r3, r19, 16
29  0x000001EC addk r3, r3, r4
30  0x000001F0 swi r3, r19, 16
31  0x000001F4 lwi r4, r19, 12
32  0x000001F8 bsrli r3, r4, 31
33  0x000001FC addk r3, r3, r4
34  0x00000200 sra r3, r3
35  0x00000204 swi r3, r19, 12
36  0x00000208 lwi r3, r19, 20
37  0x0000020C addik r3, r3, 1
38  0x00000210 swi r3, r19, 20
39  0x00000214 lwi r4, r19, 20
40  0x00000218 lwi r3, r19, 4
41  0x0000021C cmp r18, r3, r4
42  0x00000220 blti r18, -64
43  0x000001E0 lwi r3, r19, 12
44  0x000001E4 andi r4, r3, 1
45  0x000001E8 lwi r3, r19, 16
46  0x000001EC addk r3, r3, r4
47  0x000001F0 swi r3, r19, 16
48  0x000001F4 lwi r4, r19, 12
49  0x000001F8 bsrli r3, r4, 31
50  0x000001FC addk r3, r3, r4
51  0x00000200 sra r3, r3
52  0x00000204 swi r3, r19, 12
53  0x00000208 lwi r3, r19, 20
54  0x0000020C addik r3, r3, 1
55  0x00000210 swi r3, r19, 20
56  0x00000214 lwi r4, r19, 20
57  0x00000218 lwi r3, r19, 4
58  0x0000021C cmp r18, r3, r4
59  0x00000220 blti r18, -64
60  0x000001E0 lwi r3, r19, 12
61  0x000001E4 andi r4, r3, 1
62  0x000001E8 lwi r3, r19, 16
63  0x000001EC addk r3, r3, r4
64  0x000001F0 swi r3, r19, 16
65  0x000001F4 lwi r4, r19, 12
66  0x000001F8 bsrli r3, r4, 31
67  0x000001FC addk r3, r3, r4
68  0x00000200 sra r3, r3
69  0x00000204 swi r3, r19, 12
70  0x00000208 lwi r3, r19, 20
71  0x0000020C addik r3, r3, 1
72  0x00000210 swi r3, r19, 20
73  0x00000214 lwi r4, r19, 20
74  0x00000218 lwi r3, r19, 4
75  0x0000021C cmp r18, r3, r4
76  0x00000220 blti r18, -64
77  0x000001E0 lwi r3, r19, 12
78  0x000001E4 andi r4, r3, 1
79  0x000001E8 lwi r3, r19, 16
80  0x000001EC addk r3, r3, r4
81  0x000001F0 swi r3, r19, 16
82  0x000001F4 lwi r4, r19, 12
83  0x000001F8 bsrli r3, r4, 31
84  0x000001FC addk r3, r3, r4
85  0x00000200 sra r3, r3
86  0x00000204 swi r3, r19, 12
87  0x00000208 lwi r3, r19, 20
88  0x0000020C addik r3, r3, 1
89  0x00000210 swi r3, r19, 20
90  0x00000214 lwi r4, r19, 20
91  0x00000218 lwi r3, r19, 4
92  0x0000021C cmp r18, r3, r4
93  0x00000220 blti r18, -64
94  0x000001E0 lwi r3, r19, 12
95  0x000001E4 andi r4, r3, 1
96  0x000001E8 lwi r3, r19, 16
97  0x000001EC addk r3, r3, r4
98  0x000001F0 swi r3, r19, 16
99  0x000001F4 lwi r4, r19, 12
100 0x000001F8 bsrli r3, r4, 31
```

Figura A.1: Parte 1: Megabloco 0x000001CC, constituido por 9 blocos básicos, obtido no benchmark pop_cnt_8bit00

```
0x000001FC addk r3, r3, r4
0x00000200 sra r3, r3
0x00000204 swi r3, r19, 12
0x00000208 lwi r3, r19, 20
0x0000020C addik r3, r3, 1
0x00000210 swi r3, r19, 20
0x00000214 lwi r4, r19, 20
0x00000218 lwi r3, r19, 4
0x0000021C cmp r18, r3, r4
0x00000220 blti r18, -64
0x000001E0 lwi r3, r19, 12
0x000001E4 andi r4, r3, 1
0x000001E8 lwi r3, r19, 16
0x000001EC addk r3, r3, r4
0x000001F0 swi r3, r19, 16
0x000001F4 lwi r4, r19, 12
0x000001F8 bsrli r3, r4, 31
0x000001FC addk r3, r3, r4
0x00000200 sra r3, r3
0x00000204 swi r3, r19, 12
0x00000208 lwi r3, r19, 20
0x0000020C addik r3, r3, 1
0x00000210 swi r3, r19, 20
0x00000214 lwi r4, r19, 20
0x00000218 lwi r3, r19, 4
0x0000021C cmp r18, r3, r4
0x00000220 blti r18, -64
0x000001E0 lwi r3, r19, 12
0x000001E4 andi r4, r3, 1
0x000001E8 lwi r3, r19, 16
0x000001EC addk r3, r3, r4
0x000001F0 swi r3, r19, 16
0x000001F4 lwi r4, r19, 12
0x000001F8 bsrli r3, r4, 31
0x000001FC addk r3, r3, r4
0x00000200 sra r3, r3
0x00000204 swi r3, r19, 12
0x00000208 lwi r3, r19, 20
0x0000020C addik r3, r3, 1
0x00000210 swi r3, r19, 20
0x00000214 lwi r4, r19, 20
0x00000218 lwi r3, r19, 4
0x0000021C cmp r18, r3, r4
0x00000220 blti r18, -64
0x00000224 lwi r4, r19, 8
0x00000228 lwi r3, r19, 16
0x0000022C addk r3, r4, r3
0x00000230 swi r3, r19, 8
0x00000234 lwi r3, r19, 24
0x00000238 addik r3, r3, 1
0x00000208 lwi r3, r19, 20
0x0000020C addik r3, r3, 1
0x00000210 swi r3, r19, 20
0x00000214 lwi r4, r19, 20
0x00000218 lwi r3, r19, 4
0x0000021C cmp r18, r3, r4
0x00000220 blti r18, -64
0x00000224 lwi r4, r19, 8
0x00000228 lwi r3, r19, 16
0x0000022C addk r3, r4, r3
0x00000230 swi r3, r19, 8
0x00000234 lwi r3, r19, 24
0x00000238 addik r3, r3, 1
0x0000023C swi r3, r19, 24
0x00000240 lwi r3, r19, 24
0x00000244 addik r18, r0, 1023
0x00000248 cmp r18, r3, r18
0x0000024C bgei r18, -128      // fim do megabloco, instrução de branch
```

Figura A.2: Parte 2: Megabloco 0x000001CC, constítuido por 9 blocos básicos, obtido no benchmark pop_cnt_8bit00

Anexo B

Documentação do módulo Megablock Extractor

B.1 Sistema

Este sistema tem como objetivo encontrar Megablocos, padrões encontrados em partes do código em execução formado por blocos básicos num programa a correr no Microblaze *soft processor* e guardar a informação necessária sobre os mesmos. O intuito do sistema (fig. B.1) vem da necessidade de acelerar sistemas embebidos e uma das técnicas usadas emprega para informação sobre os ciclos executadas pelo processador, os Megablocos que este contém durante o sua execução tal como estatísticas sobre os mesmos. Constituído por um detetor de blocos básicos e a seguir um detetor de *squares* (repetição de um padrão duas vezes), a implementação encontra estes Megablocos e guarda-os em memória.

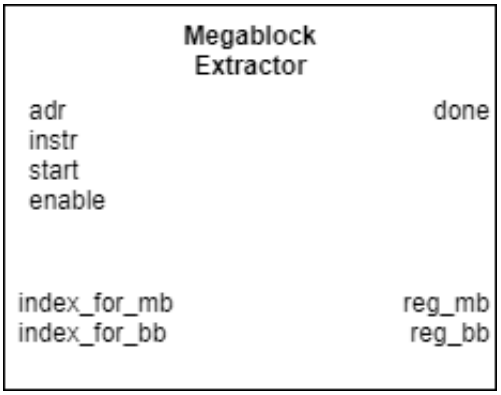

Figura B.1: Módulo Megablock Extractor

B.1.1 Funcionalidades

Como dito anteriormente, o sistema encontra Megablocos, formados por blocos básicos, e informações sobre os mesmos. Dessas informações as mais relevantes são os blocos básicos indi-

Tabela B.1: Interface do módulo

Pin	Tipo	Bits	Descrição
adr	input	32	Endereço das instruções do soft processor
instr	input	7	Bits necessários das instruções do soft processor
start	input	1	Indica primeira início do primeiro bloco básico
enable	input	1	Enquanto ativo indica que o sistema continua a procura de padrões
big_pattern	input	1	Decide entre detetar loops desenrolados ou não
index_for_bb	output	16	Índice usado para retirar os dados de reg_bb
index_for_mb	output	16	Índice usado para retirar os dados de reg_mb
done	output	1	Indica que é possível retirar os valores das memórias
reg_mb	output	76	Sinais com os valores da memória com as estatisticas de cada Megabloco
reg_bb	output	64	Sinais com os valores da memória com os blocos básicos pertencentes a cada padrão

viduais que formam cada Megabloco. Sobre cada Megabloco é guardado o número de iterações máximas e totais; no caso de ocorrências intermitentes é guardada a maior. Também sobre cada um é registado o tamanho do Megabloco contado em número de blocos básicos. Além destas, informações é guardado o número de ocorrências do bloco básico, por exemplo ababxxabab são duas ocorrências consecutivas do Megabloco ab.

O sistema funciona à frequência normal do Microblaze, 100 Mhz, usando 2% dos recursos da placa Zedboard com Zynq-7020.

B.1.2 Parâmetros

Considerando o a funcionalidade deste sistema, parâmetros foram uma forma de controlar os recursos usados pelo sistema e os requisitos do mesmo. Parâmetros como MAX_MB, que indica qual o tamanho máximo desejado do Megabloco, ou TOTAL_MB e MEMORY, que indica o tamanho das memórias que guardam as informações sobre iterações de cada Megabloco e os blocos básicos correspondentes a cada Megabloco, respetivamente. Além destes existe um parâmetro que define o tamanho do contador usado nas estatísticas, COUNT_SIZE. Os outros dois definem o número de bits de um endereço e o número de bits necessário de cada instrução de acordo com instruções de *branch* Microblaze [11].

Um sinal que, apesar de não ser parâmetro, não é alterado durante a execução dos sistema é *big_pattern*. Este tem a função de definir a prioridade na deteção de padrões. Estando ativo indica prioridade a *loops* desenrolados. Por exemplo para o padrão baabaa : baa é detetado, caso contrário esse não seria detetado mas sim o padrão aa.

B.1.3 Modo de utilização

Como se pode ver pela tabela B.1 existem dois sinais importantes para o funcionamento correto do sistema. O sinal de *start* indica quando o primeiro bloco básico começa. Esta informação serve para não se perder nenhuma iteração de um Megabloco caso o *profiling* do sistema aconteça em qualquer ponto da execução. O sinal de *enable* deve ser usado quando se quer fazer *profiling*

durante alguma parte da execução. O módulo pode ser ativado a qualquer momento de execução do sistema. Após ativado o sinal de *enable*, o sistema começa detetar os padrões e guardar as informações sobre cada um deles. E após ser desativado é possível aceder as estas informações através do sinais de *index_for_mb*, *index_for_bb*, *reg_mb*, *reg_bb*. Se for ativado outra vez, as informações anteriores podem ser e atualizadas e acrescentadas com novas informações.

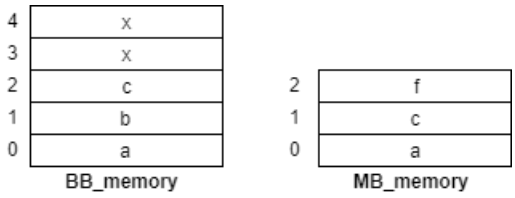

Figura B.2: Exemplo de Megablocos registados em memória simplificado. Megabloco: ab; identificado por a e Megabloco:c

Na figura B.2, é mostrado um exemplo de memória cheias ao longo da execução de um programa. Para aceder a estes valores é necessário esperar que *done* esteja ativo, sinalizando que a escrita parou. Após este sinal ativo, é necessário indicar de que endereço se quer retirar as informações através de *index_for_mb* e *index_for_bb* de *reg_mb* e *reg_bb*, respetivamente. Podem ser recebidos dados de quaisquer índice a qualquer altura. No entanto um protocolo sugerido é começar por verificar os resultados de todos os índices de MB_memory até encontrar um identificador com valor 0xffff.

A cada índice verificado retira-se o tamanho de um dos Megabloco guardado, e depois retiram-se os valores de BB_memory até ao índice do tamanho-1. Assim obtém-se a informação para formar o Megabloco. Isto é feito até se obter todos os índices de MB_memory e os blocos básicos respetivos em BB_memory. O exemplo de execução na figura B.3 mostra o processo. Embora omita alguma informação permite, ver como retirar a informação em MB_memory, tal como o tamanho do Megabloco identificado.

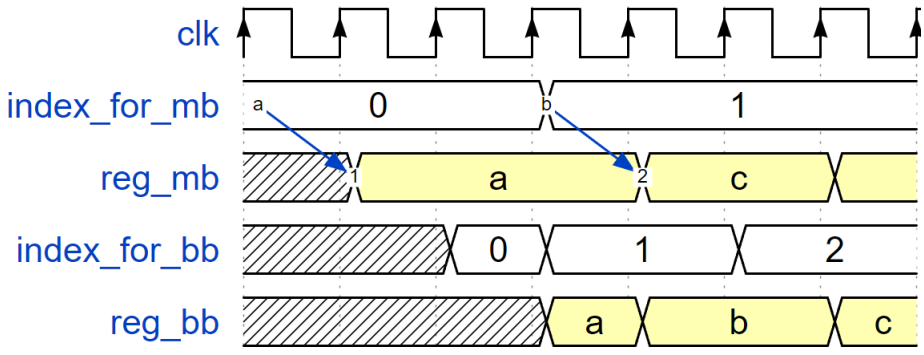

Figura B.3: Exemplo de como retirar a informação sobre cada Megabloco (em relação à figura anterior)

Referências

[1] João Bispo, Nuno Paulino, João MP Cardoso, e João Canas Ferreira. Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units, 2013.

[2] Ann Gordon-Ross e Frank Vahid. Frequent loop detection using efficient nonintrusive on-chip hardware. *IEEE Transactions on Computers*, 54(10):1203–1215, 2005.

[3] Ajay Nair e Roman Lysecky. Non-intrusive dynamic application profiler for detailed loop execution characterization. Em *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, páginas 23–30. ACM, 2008.

[4] Ajay Nair, Karthik Shankar, e Roman Lysecky. Efficient hardware-based nonintrusive dynamic application profiling. *ACM Transactions on Embedded Computing Systems (TECS)*, 10(3):32, 2011.

[5] Yukinori Sato, Ken-ichi Suzuki, e Tadao Nakamura. Run-time detection mechanism of nested call-loop structure to monitor the actual execution of codes. Em *2009 Software Technologies Future Dependable Distributed Systems, for*, páginas 184–188. IEEE, 2009.

[6] Mark Aldham, Jason Anderson, Stephen Brown, e Andrew Canis. Low-cost hardware profiling of run-time and energy in fpga embedded processors. Em *ASAP 2011-22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, páginas 61–68. IEEE, 2011.

[7] Martin Schulz, Brian S White, Sally A McKee, Hsien-Hsin S Lee, e Jürgen Jeitner. Owl: next generation system monitoring. Em *Proceedings of the 2nd Conference on Computing Frontiers*, páginas 116–124. ACM, 2005.

[8] Roman Lysecky, Susan Cotterell, e Frank Vahid. A fast on-chip profiler memory. Em *Proceedings of the 39th annual Design Automation Conference*, páginas 28–33. ACM, 2002.

[9] Frank Vahid, Greg Stitt, e Roman Lysecky. Warp processing: Dynamic translation of binaries to fpga circuits. *Computer*, 41(7), 2008.

[10] João Carlos Viegas Martins Bispo. *Mapping Runtime-Detected Loops from Microprocessors to Reconfigurable Processing Units*. Tese de doutoramento, Instituto Superior Técnico, 2012.

[11] Xilinx UG081. Microblaze processor reference guide, 2006.